

## Introdução

A linguagem Pascal foi desenvolvida pelo professor Niklaus Wirth no ano de 1972, na cidade de Genebra, Suíça. O nome da linguagem foi uma homenagem ao filósofo e matemático Blaise Pascal ( 1623-1662 ), inventor da primeira calculadora mecânica. O desejo de Wirth era dispor, para o ensino de programação, de nova linguagem que fosse simples, coerente e capaz de incentivar a confecção de programas claros e facilmente legíveis, favorecendo a utilização de boas técnicas de programação.

A linguagem Pascal se tornou amplamente conhecida e utilizada com o lançamento da mundialmente famosa série de compiladores Turbo Pascal pela Borland, em 1985, devido à uma combinação de simplicidade e poder.

O compilador Pascalzim, desenvolvido no Departamento de Ciências da Computação da Universidade de Brasília, é fruto de muitos anos de pesquisa e trabalho na área de tradutores e linguagens de programação. Adotado como ferramenta de apoio ao ensino e aprendizagem da linguagem Pascal pelos alunos matriculados no curso de Introdução à Ciência da Computação nesta instituição, o compilador foi utilizada no primeiro semestre do ano 2000.

No segundo semestre de 2001 a ferramenta foi utilizada pelos alunos do Instituto de Ensino Superior de Brasília - IESB para o aprendizado da disciplina Algoritmos e Programação Estruturada.

A ferramenta foi exaustivamente testada em laboratório, mas ainda assim não se encontra livre de erros de implementação. A correção destes será efetuada tão breve quanto sua descoberta, levando à disponibilidade de versões mais atualizadas do compilador.

O compilador implementa um subconjunto da linguagem Pascal e contém as estruturas de dados, funções e comandos mais utilizados por iniciantes no estudo dessa linguagem. O arquivo de ajuda que acompanha o produto especifica as instruções suportadas.

Críticas e sugestões são bem vindas!

O Pascalzim foi concebido com finalidade meramente educacional e sua distribuição é livre.

---

*Created with the Freeware Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)*

## Formato Básico Programa

Um programa escrito na linguagem **Pascal** pode ser, basicamente, estruturado em três regiões significativas:

1. Um cabeçalho, que dá nome ao algoritmo;
2. Uma seção de definição e declaração de dados;
3. Uma seção de comandos, que contém as instruções do programa.

O cabeçalho de um programa é iniciado com a palavra reservada **Program**, seguido de um nome que identifica o programa e um ponto e vírgula.

### Exemplo

```
Program MeuPrograma ;
```

A seção de definição e declaração de dados segue o cabeçalho do programa, e é o local onde são definidas as constantes e tipos que serão usados dentro do programa. Nesta seção também são declaradas as variáveis globais do programa, assim como as funções e procedimentos que podem ser utilizados pelo programa principal.

Essa seção consiste das seguintes partes:

1. A parte para [declaração de constantes](#);
2. A parte para [definição de tipos](#);
3. A parte para [declaração de variáveis](#);
4. A parte para [definição de funções e procedimentos](#);

A definição de cada uma dessas partes é opcional, mas deve seguir a ordem estabelecida. Por exemplo, uma função não pode ser definida antes da declaração de uma variável.

Em seguida, deve ser elaborada a [seção de comandos](#). Esta seção é iniciada com a palavra reservada **Begin** e terminada com a palavra reservada **End**, seguida de ponto. Entre as palavras **Begin** e **End** devem ser colocados os comandos do programa.

Seguindo essa ideia, o formato genérico de um programa Pascal tem a seguinte estrutura:

```
Program NomePrograma ;  
  
Seção de definições e declarações  
  
Begin  
  Comandos  
End .
```

onde **Program**, **Begin** e **End** são [palavras reservadas](#) da linguagem Pascal.

## Identificadores

Um identificador válido na linguagem **Pascal** é qualquer seqüência de caracteres que obedeça às seguintes regras:

1. Seja iniciada por uma letra ( a, b, ..., z ) ;
2. Possui, depois da primeira letra, uma seqüência de caracteres que podem ser letras, dígitos (1, 2, ... , 9, 0 ) ou ainda o caractere \_ ;
3. É uma palavra que não é uma das [palavras reservadas](#) da linguagem Pascal.

### Exemplo

Identificadores válidos na linguagem Pascal:

**A**  
**Nota**  
**P1**  
**Meu\_Identificador**

Identificadores inválidos na linguagem Pascal:

**1A**  
**E(13)**  
**A:B**

A linguagem Pascal não diferencia palavras maiúsculas de palavras minúsculas. Assim, para o compilador as seguintes palavras denotam um mesmo identificador:

PASCAL = pascal = Pascal

## Palavras Reservadas

O conjunto das *palavras reservadas* da linguagem Pascal, identificadas pelo compilador é dado pela tabela abaixo:

APPEND	TEXTCOLOR
ARRAY	CHR
ASSIGN	RED
BEGIN	:
BOOLEAN	YELLOW
CHAR	LIGHTCYAN
CLOSE	LIGHTGREEN
CLRSCR	;
CONST	LENGTH
DO	TEXTBACKGROUND
DOWTO	BLINK
ELSE	>
END	NOT
FALSE	LIGHTGRAY
FOR	GREEN
FUNCTION	/
GOTOXY	LIGHTMAGENTA
IF	TEXT
INTEGER	MOD
OF	LIGHTBLUE
ORD	MAGENTA
PROCEDURE	*
PROGRAM	OR
READ	.
READKEY	[
READLN	-
REAL	BROWN
RECORD	(
REPEAT	,
RESET	=
REWRITE	DARKGRAY
STRING	AND
THEN	<
TO	BLUE
TRUE	CYAN
TYPE	)
UNTIL	LIGHTRED
VAR	WHITE
WHILE	DIV
WRITE	]

WRITELN

EOF

---

Created with the Freeware Edition of HelpNDoc: [Easy CHM and documentation editor](#)

## ConstantesPreDefinidas

O compilador reconhece as seguintes constantes pré-definidas:

- ***maxint***

Guarda o valor máximo de um inteiro, 32.767

- ***pi***

Guarda o valor da constante pi, 3.14159265358979

## Declaração de Constantes

As constantes são declaradas na seção de declaração de constantes, que está contida na seção de definição e declaração de dados.

O início da seção de declaração de constantes é indicada por meio da palavra reservada **const**. A palavra reservada **const** marca o início da seção de definições de constantes, e deve aparecer somente uma única vez dentro da seção de declarações e definições.

### Sintaxe

**const**

*Identificador1, identificador2, .... , identificador<sub>n</sub> = constante ;*

onde *constante* deve ser uma constante inteira, real, uma cadeia de caracteres ou um único caractere.

Exemplo. A declaração abaixo define uma constante inteira cujo valor é 10:

```
const dez = 10 ;
```

## Tipos de Dados

Toda as variáveis declaradas dentro de um programa devem ser especificadas através de um *tipo*.

Um *tipo* é uma especificação que:

- Indica o espaço em memória necessário para o armazenamento de um dado (ou conjunto de dados)
- Define o conjunto de operações que pode ser aplicada a um dado (ou conjunto de dados)

Os tipos implementados no compilador podem, basicamente, serem classificados em três categorias:

1. [Tipos predefinidos](#)
2. [Tipos estruturados](#)
3. [Tipos definidos](#)

## Tipos Pré-Definidos

Os tipos de dados predefinidos na linguagem **Pascal**, e implementados no compilador, são:

- **Boolean**

- Define dois valores lógicos: FALSE e TRUE.
- Um dado do tipo booleano ocupa um byte de espaço na memória.

- **Char**

- Define os elementos do conjunto de caracteres que compõem o alfabeto ASCII, adicionados dos caracteres representados pelos códigos de 128 a 255.

- Um dado do tipo char ocupa um byte de espaço na memória.

- **Integer**

- Define os valores inteiros compreendidos no intervalo de -2.147.483.647 até 2.147.483.647.

- Um dado do tipo integer ocupa quatro bytes de espaço na memória.

- **Real**

- Define os valores reais definidos no intervalo de  $3.4 \times (10^{-38})$  até  $3.4 \times (10^{+38})$ .

- Um dado do tipo real ocupa quatro bytes de espaço na memória.

- **String**

- Define uma cadeia de caracteres. Se nenhuma restrição de tamanho for especificada, um dado do tipo string é capaz de armazenar uma sequência contendo até 255 caracteres, onde cada caracter ocupa um byte de espaço na memória.

Uma cadeia de caracteres pode ter seu tamanho definido (contendo menos de 255 caracteres), onde o tamanho especifica o número máximo de caracteres contidos na cadeia. Essa especificação deve ser indicada entre colchetes, logo após a palavra reservada string,

### Exemplo

string [6] define uma cadeia capaz de armazenar até 6 caracteres.

Uma cadeia de caracteres definida com  $n$  caracteres ocupa  $n$  bytes de espaço na memória

## Tipos Estruturados

Os tipos de dados predefinidos podem ser organizados em tipos de dados complexos, denominados *tipos estruturados*.

A linguagem Pascal oferece, basicamente, quatro destes tipos:

1. [Enumerações](#)
2. [Ponteiros](#)
3. [Registros](#)
4. [Vetores](#)

# Enumerações

Tipos de dados enumerados são utilizados para denotar um conjunto de constantes.

## Declaração de enumerações

```
Var nomeEnumeracao : ( identificador, ....., identificador ) ;
```

Onde identificador denota um identificador válido na linguagem Pascal.

## Exemplo

```
Program Pzim ;  
var diasSemana : (domingo, segunda, terca, quarta, quinta, sexta, sabado) ;  
Begin  
  writeln( 'Depois de segunda vem quinta?' , succ(segunda) = quinta );  
  writeln( 'Depois de segunda vem terca?' , succ(segunda) = terca );  
  writeln( 'Antes de quinta vem quarta?' , pred(quinta) = quarta );  
  writeln( 'Antes de quinta vem segunda?' , pred(quinta) = segunda );  
End.
```

## Exemplo

```
Program Pzim ;  
Type diaSemana = ( domingo, segunda, terca, quarta, quinta, sexta,  
sabado ) ;  
Var dia : diaSemana ;  
Begin  
  for dia := domingo to sabado do  
    begin  
      case ( dia ) of  
        domingo: writeln( '0 dia é domingo' );  
        segunda: writeln( '0 dia é segunda' ) ;  
        terca : writeln( '0 dia é terca' ) ;  
        quarta : writeln( '0 dia é quarta' ) ;  
        quinta : writeln( '0 dia é quinta' ) ;  
        sexta : writeln( '0 dia é sexta' ) ;  
        sabado : writeln( '0 dia é sabado' ) ;  
      end;
```

```
end;  
readkey;  
End.
```

---

Created with the Freeware Edition of HelpNDoc: [Free help authoring environment](#)

# Ponteiros

Ponteiros são variáveis que podem armazenar o endereço de uma outra variável.

## Declaração de ponteiros

```
Var nomePonteiro : ^tipoDados ;
```

O símbolo ^ deve ser lido como o *ponteiro para...*

Na declaração acima temos que *nomePonteiro* é um ponteiro para variáveis do tipo *tipoDados*.

## Exemplo

```
Var ponteiro: ^integer ;
```

## Exemplo

```
Type TAluno = Record  
    nome: String ;  
    matricula: String ;  
End ;
```

```
Var ponteiroAluno: ^TAluno ;
```

## Operações sobre ponteiros

- Guardar no ponteiro endereço de uma variável:

```
ponteiro := @variável ;
```

- Guardar no ponteiro o endereço armazenado em um outro ponteiro:

```
ponteiro := outroponteiro ;
```

- Dizer que o ponteiro não guarda nenhum endereço:

```
ponteiro := nil ;
```

- Referenciar o dado apontado pelo ponteiro (o elemento que tem o tipo de dados definido pelo ponteiro, e que está no endereço de memória que o ponteiro armazena):

```
ponteiro^
```

## Exemplo

```
Program Ponteiros ;  
Var a: integer;  
    p: ^integer;  
Begin  
    a := 8 ;    // Guardamos o valor 8 em a  
    p := nil;  // O ponteiro não guarda nenhum endereço  
    writeln( 'Valor armazenado em a: ' , a );  
    // Guardamos no ponteiro o endereço da variável a  
    p := @a ;  
    writeln( 'Valor apontado por p: ' , p^ );  
    // Esse comando é equivalente a "a:= 2 * a ;" , pois p  
    // aponta para o endereço de a  
    a:= 2 * p^ ;  
    writeln( 'O valor de a agora: ' , a );    // Imprime 16  
    writeln( 'Valor apontado por p: ' , p^ ); // Imprime 16  
    readln ;  
End.
```

## Alocação Dinâmica de Memória

É possível alocar, dinamicamente, espaço na memória para um ponteiro.  
A quantidade de memória é determinada pelo tipo do ponteiro.

### Sintaxe

```
new( ponteiro ) ;
```

Deve-se tomar cuidado para que a memória alocada com um *new* seja liberada antes do programa terminar.

### Sintaxe

```
dispose( ponteiro ) ;
```

## Exemplo

```
Program AlocacaoDinamica ;  
Var p: ^integer;  
    v : integer ;  
Begin  
    new( p ); // Aloca espaço para armazenar um inteiro  
    p^ := 10 ; // Guarda um inteiro na posição apontada por p  
  
    writeln( 'Valor armazenado na posicao de memoria: ', p^ );  
  
    v:= p^ ; //Guardamos em v o valor apontado por p  
  
    writeln( 'Valor armazenado em v: ', v );  
  
    dispose( p ); // Liberamos a memoria associada a p  
    readln ;  
End.
```

## Exemplo

```
// -----  
// Este programa mostra ilustra a utilização de listas lineares  
// usando ponteiros.  
//  
// Problema. Construir uma lista linear e imprimir seus dados.  
// -----
```

```
Program PercorrendoLista ;
```

```
// Definição de um tipo para representar um nó da lista
```

```
type TNo = record
```

```
    dado : integer ; // Dado armazenado pelo nó
```

```
    prox : ^TNo ; // Ponteiro p/ próximo nó
```

```
end ;
```

```
Var pinicio: ^TNo; // Guarda endereço 1º nó da lista
```

```
    p1: ^TNo; // Auxiliar. Guarda endereço de um nó
```

```
    resposta : char ; // Auxiliar. Controla repetição.
```

```
Begin
```

```
    pinicio := nil ;
```

```
// Repetição que define os nós da lista
```

```
repeat
```

```
    new( p1 );
```

```
    write( 'Entre com novo dado: ' );
```

```
    readln( p1^.dado ) ;
```

```
    p1^.prox := pinicio ;
```

```
    pinicio := p1 ;
```

```
    write( 'Novo dado(S/N)?' );
```

```
    readln( resposta );
```

```
    resposta := upcase( resposta );
```

```
Until resposta = 'N' ;
```

```
// Percorrer a lista, imprimindo seus elementos
```

```
p1 := pinicio ;
```

```
while( p1 <> nil ) do
```

```
Begin
  writeln( 'Achei: ' , p1^.dado );
  p1 := p1^.prox ;
End;

// Percorrer a lista, desalocando memória para os elementos
while( pinicio <> nil ) do
Begin
  p1 := pinicio ;
  pinicio := pinicio^.prox ;
  dispose( p1 );
End;

  readln ;
End.
```

# Registros

Um *registro* é um tipo composto por um conjunto formado por dados de tipos diferentes, onde cada um dados é definido como sendo um *campo*. Um tipo *registro* é definido através da palavra reservada **record**, seguida por uma série de declaração de *campos*. A palavra reservada **end** seguida de um ponto e vírgula encerra a definição de um registro.

A sintaxe genérica para definição de *registros* segue o seguinte formato:

## Sintaxe

### **Record**

```
Identificador de campo : tipo;  
Identificador de campo : tipo;  
  
Identificador de campo : tipo;  
End;
```

Exemplo. Declaração de um registro simples:

```
var dados : Record  
    numero : integer;  
    caracter : char;  
    preenchido : boolean;  
End;
```

Exemplo. Declaração de um registro contendo registros aninhados:

```
var umRegistro : Record  
    numero : integer ;  
    dado : Record  
        caractere : char ;  
        End;  
    preenchido: boolean ;  
End;
```

A referência a um campo de um registro é feita através do nome da variável do tipo registro seguida por um ponto e pelo nome do campo, como por exemplo,

umRegistro.numero

---

Created with the Freeware Edition of HelpNDoc: [Free help authoring environment](#)

## Vetores

Um vetor é uma estrutura de dados que contém um número fixo de elementos que possuem um mesmo tipo de dados, tipo esse que pode ser qualquer um dos tipos predefinidos na linguagem Pascal ( integer, char, boolean ou string ), um tipo *vetor*, um tipo *registro* ou ainda um tipo definido pelo usuário.

O número de elementos de um vetor é determinado pelo intervalo de indexação de elementos do vetor, que é especificado por duas constantes ordinais separadas por dois pontos, entre colchetes.

A sintaxe para definição de vetores segue o seguinte formato:

```
array[ limiteInferior .. limiteSuperior ] of tipo ;
```

Onde:

- **array** e **of** são palavras reservadas da linguagem Pascal, usadas na declaração de vetores
- limiteInferior e limiteSuperior são constantes ordinais;
- tipo define o tipo de dados de cada elemento do vetor

Exemplo. A declaração abaixo define um vetor do tipo inteiro, identificado por *Dias*:

```
Var  
  dias : array [ 1 .. 24 ] of integer;
```

Nesse vetor, os elementos estão armazenados nas posições de 1 a 24.

A referência ao elemento que está armazenado na posição *x* de um vetor é dado da seguinte forma:

```
nomeVariavel[ x ]
```

Os vetores podem ter, ainda, [mais de uma dimensão](#).

---

*Created with the Freeware Edition of HelpNDoc: [Create HTML Help, DOC, PDF and print manuals from 1 single source](#)*

## Vetores Com Várias Dimensões

Vetores podem ter mais de uma dimensão. Nesse caso, cada nova dimensão é declarada de acordo com as regras do item anterior, e as  $n$  dimensões do vetor são separadas por vírgulas.

A sintaxe para definição vetores  $n$ -dimensionais segue o seguinte formato:

```
array[ limite1 .. limite2 , limite3 .. limite4 , ... , limite $n-1$  ..  
limite $n$  ] of tipo;
```

Exemplo. A declaração abaixo define um vetor de duas dimensões, do tipo inteiro:

```
var  
matriz : array [1 .. 10, 1.. 20] of integer ;
```

## Definicao de Tipos

A definição de um novo tipo é feita na seção de definição de tipos, contida na seção de definição e declaração de dados.

O início da seção de definição de tipos é indicada através da palavra reservada **Type**. A palavra reservada **Type** deve aparecer uma única vez dentro da seção de definição e declaração de dados.

### Sintaxe

**type**

```
nomeTipo = tipoDefinido ;
```

onde tipoDefinido é um dos tipos estruturados *vetor*, *registro*, *ponteiro* ou outro tipo de dados simples.

### Exemplo

**type**

```
intList = array[1..100] of integer ;  
matrix = array[0..9, 0..9] of real ;  
pInt = ^integer ;
```

## Declaração de Variáveis

A declaração de uma variável faz com que o compilador reserve uma quantidade de espaço em memória suficientemente grande para armazenar um tipo de dados, além de associar também um “nome” a esta posição de memória. As variáveis são declaradas na seção de declaração de variáveis, contida na seção de definição e declaração de dados.

O início da seção de declaração de variáveis é indicada por meio da palavra reservada **Var**. A palavra reservada **Var** deve aparecer somente uma única vez dentro da seção de definição e declaração de dados.

### Sintaxe

```
var  
    identificador1, identificador2, .... , identificadorn : tipo ;
```

Exemplo. A declaração abaixo define três variáveis dos tipos inteiro, caractere e booleano, respectivamente..

```
var  
    inteiro: integer;  
    caractere: char;  
    booleano: boolean;
```

## Expressões

O termo *expressão* se refere a qualquer combinação de uma ou mais constantes ou identificadores de variáveis, com um ou mais *operadores*. As constantes e variáveis que aparecem numa expressão são chamadas de *operandos*.

Quando mais de um operador aparece numa expressão, a seqüência de cálculo efetuada pelo compilador depende da precedência definida para cada operador da linguagem, onde o operador com mais alta precedência é o primeiro a capturar seus operandos. No caso de dois ou mais operadores terem o mesmo nível de precedência, o cálculo é feito da esquerda para a direita.

São definidos quatro níveis de precedência para os operadores da linguagem, definidos abaixo em ordem decrescente:

1. - (menos unário), not
2. \*, div, mod, and
3. +, -, or
4. =, <>, <, >, <=, >=

Parênteses alteram a ordem de precedência de um conjunto de operadores, forçando o programa a calcular a expressão dentro dos parênteses antes das outras.

Por exemplo, a adição é calculada antes da multiplicação em  $5 * (3 + 4)$ .

# Operadores

Grande parte da manipulação de dados que ocorre na seção de comandos é feita através pelo uso de um *operador*.

Um *operador*, na linguagem Pascal, pertence basicamente a uma dentre as três categorias básicas abaixo:

1. [operadores aritméticos](#)
2. [operadores lógicos](#)
3. [operadores relacionais](#)

## Operadores Aritméticos

Os operadores aritméticos são utilizados nas expressões aritméticas. Os operadores aritméticos definidos pelo compilador são:

- - (Menos Unário)  
Tipo de operando permitido: inteiro ou real.  
Operação executada: Inverte o valor numérico do operando.
  
- **DIV**  
Tipo de operandos permitidos: ambos do tipo inteiro.  
Operação executada: O operando à esquerda do DIV é dividido pelo operando à sua direita, sendo o resultado desta operação um valor inteiro resultante da divisão.
  
- **MOD**  
Tipo de operandos permitidos: ambos do tipo inteiro.  
Operação executada: O operando à esquerda do MOD é dividido pelo operando à sua direita, sendo o resultado desta operação o resto inteiro da divisão.
  
- **+**  
Tipo de operandos permitidos: inteiros, reais, cadeias de caracteres.  
Operação executada: No caso de inteiros e reais o operando à esquerda do + é somado ao operando a sua direita, sendo o tipo do resultado dessa operação dependente de seus operandos:
  - Se os dois operandos são inteiros, o resultado da soma é um valor inteiro.
  - Se os dois operandos são reais, o resultado da soma é um valor real.
  - Se os um dos operandos é real, e o outro é inteiro, o resultado da soma é um valor real.

No caso dos operandos serem ambos cadeias de caracteres o resultado da operação é dada pela cadeia obtida pela concatenação da cadeia dada pelo primeiro operando com a cadeia dada pelo segundo operando.

- -

Tipo de operandos permitidos: inteiros, reais.

Operação executada: O operando à esquerda do - é subtraído do operando a sua direita, sendo o tipo do resultado dessa operação dependente de seus operandos:

- Se os dois operandos são inteiros, o resultado da operação é inteiro.
- Se os dois operandos são reais, o resultado da operação é real.
- Se os um dos operandos é real, e o outro é inteiro, o resultado da operação é real.

- \*

Tipo de operandos permitidos: inteiros, reais.

Operação executada: O operando à esquerda do \* é multiplicado pelo operando a sua direita, sendo o tipo do resultado dessa operação dependente de seus operandos:

- Se os dois operandos são inteiros, o resultado da operação é um valor inteiro.
- Se os dois operandos são reais, o resultado da operação é um valor real.
- Se os um dos operandos é real, e o outro é inteiro, o resultado da operação é um valor real.

- /

Tipo de operandos permitidos: inteiros, reais.

Operação executada: O operando à esquerda do / é dividido pelo operando a sua direita, sendo o resultado dessa operação real.

## Operadores Lógicos

Os operadores lógicos são usados nas expressões lógicas, com operandos do tipo booleano.

Os operadores lógicos definidos pelo compilador são:

- **not**

Operação executada: O operador inverte o valor verdade de um operando booleano.

- **and**

Operação executada: É efetuado um and lógico entre os dois operando do operador, sendo o resultado da operação verdadeiro quando ambos operandos são verdadeiros.

- **or**

Operação executada: É feito um or lógico entre os dois operando do operador, sendo o resultado da operação verdadeiro se um dos operandos for verdadeiro.

- **xor**

Operação executada: É feito um xor lógico entre os dois operando do operador, sendo o resultado da operação verdadeiro se os dois operandos contiverem valores lógicos diferentes.

A tabela verdade para os operadores lógicos é:

Primeiro Operando	Operador	Segundo Operando	Resultado
-------------------	----------	------------------	-----------

True	Not	----	False
------	-----	------	-------

False	Not	----	True
-------	-----	------	------

True	And	True	True
------	-----	------	------

True	And	False	False
False	And	True	False
False	And	False	False

True	Or	True	True
True	Or	False	True
False	Or	True	True
False	Or	False	False

True	Xor	True	False
True	Xor	False	True
False	Xor	True	True
False	Xor	False	False

Expressões contendo os operadores AND e OR são avaliadas em curto-circuito, a exemplo do compilador Turbo Pascal, da Borland. Dessa forma, se o primeiro operando do AND for avaliado em false, o segundo operando não é analisado. Também, se o primeiro operando do OR for avaliado em true, o segundo operando não é analisado.

## Operadores Relacionais

Os operadores relacionais são usados nas expressões relacionais.  
Os tipos de operandos permitidos para esse tipo de operadores são:

- Ambos operandos do mesmo tipo primitivo ( integer, char, boolean, char ou string )
- Operandos de tipos diferentes, onde:
  - Um operando é do tipo integer e outro do tipo real
  - Um operando é do tipo string e outro do tipo char

O resultado de expressões contendo operadores relacionais é um valor booleano, definido de acordo com a tabela a seguir.

<b>Operador</b>	<b>Resultado</b>
=	Verdadeiro se os dois operandos para o operador forem iguais Falso em caso contrário.
<>	Verdadeiro se os dois operandos para o operador forem diferentes. Falso em caso contrário.
<	Verdadeiro se o operando à esquerda do operador for menor que o operando à direita Falso em caso contrário.
<=	Verdadeiro se o operando à esquerda do operador for menor ou igual o operando à direita Falso em caso contrário
>	Verdadeiro se o operando à esquerda do operador for maior do que o operando à direita Falso em caso contrário.
>=	Verdadeiro se o operando à esquerda do operador for maior ou igual que o operando à

direita  
Falso em caso contrário.

# Comandos

Os comandos são inseridos na seção de comandos e podem ser, basicamente, classificados em sete categorias:

1. [Comandos de atribuição](#)
2. [Comandos compostos](#)
3. [Comandos de entrada e saída](#)
4. [Comandos condicionais](#)
5. [Comandos de repetição](#)
6. [Comandos para tratamento de arquivos](#)
7. [Comandos auxiliares](#)

O ponto e vírgula é usado na linguagem *Pascal* como separador de comandos, servindo para separar um comando dos comandos subsequentes.

## ComandosAtribuicao

Um comando de atribuição é definido através da seguinte sintaxe:

variável := expressão ;

O tipo da expressão deve ser igual ao tipo da variável, com exceção de dois casos especiais onde:

- A variável é do tipo **real** e a expressão é do tipo **integer**
- A variável é do tipo **string** e a expressão é do tipo **char**

Exemplo. Sendo dados :

```
Var
item: integer;
saida: boolean;
soma, Valor: real;
caractere: char;
cadeia: string
```

São válidos os seguintes comandos de atribuição:

```
item:= 0 ;
saida:= FALSE ;
soma:= valor1 + valor2 ;
caracter:= 'a' ;
cadeia:= 'Isso é uma cadeia de caracteres' ;
soma:= 9 ;
cadeia:= 'a' ;
```

## Comandos Compostos

Além de marcar o início e o fim da seção de comandos, o par **begin** e **end** define um par de instruções usado para combinar um conjunto de comandos em um *comando composto*, também chamado de *bloco de comandos*..

### Exemplo

```
if first < last then
  begin
    Temp := First;
    First := Last;
    Last := Temp;
  end;
```

## ComandosEntradaSaida

Os comandos usados para entrada e saída de dados são definidos, pelo compilador, por meio de quatro comandos:

1. [Read](#)
2. [Readln](#)
3. [Write](#)
4. [Writeln](#)

## read

Os comandos **read** e **readln** são usados para ler o valor de uma variável de um dispositivo de entrada de dados. A diferença entre os dois comandos é que o comando **readln** processa uma quebra de linha após a leitura do valor de uma variável, enquanto o **read** não o faz.

A leitura de dados pode ser direcionada para um arquivo, identificado por uma variável do tipo TEXT.

Sintaxe:

```
read( listaVariáveis );
```

Onde listaVariáveis é uma sequência de uma ou mais variáveis separadas por vírgula.

A sintaxe de um comando **read** para leitura a partir de um arquivo é:

```
read( variavelArquivo, listaVariáveis );
```

Onde variavelArquivo é uma variável do tipo TEXT.

### Exemplo

```
Program PascalZIM ;
var
  arq: Text ;
  caractere: char ;
begin
  assign( arq, 'Teste.Pas' );
  reset( arq );
  while not eof( arq ) do
    begin
      read( arq, caractere );
      write( caractere );
    end;
end.
```



## write

Os comandos **write** e **writeln** são usados para imprimir o valor de uma sequência de expressões em um dispositivo de saída de dados. A diferença entre os dois comandos é que o comando **writeln** processa uma quebra de linha após imprimir o valor de uma sequência de expressões.

A escrita de dados pode ser direcionada para um arquivo, identificado através de uma variável do tipo TEXT.

A sintaxe de um comando **write** / **writeln** para impressão na tela de uma sequência de expressões é:

```
write( expressão1 , expressão2 , .... , expressão ) ;
```

A sintaxe de um comando **write** / **writeln** para impressão em arquivo de uma sequência de expressões é:

```
write( variavelArquivo, expressão1 , expressão2 , .... , expressão ) ;
```

Onde variavelArquivo é uma variável do tipo TEXT.

### Exemplo

```
Program PascalZIM ;
var
  c: char ;
begin
  writeln( 'Please press a key' );
  c := Readkey;
  writeln( ' Você pressionou ', c, ', cujo valor ASCII é ', ord(c),
  '.' ) ;
end.
```

Os parâmetros do comando **write** podem conter a especificação do seu comprimento. Tal especificação é definida através da seguinte regra

sintática:

expressão : tamanho

Onde expressão define um parâmetro e *tamanho* é uma expressão do tipo inteiro.

A impressão de constantes em ponto flutuante pode conter, além da especificação de comprimento, a especificação do número de casas decimais a serem impressas. Essa especificação é dada através da seguinte regra sintática:

expressão : tamanho : casas decimais

Onde expressão é um parâmetro do tipo real, tamanho e casas decimais são expressões do tipo inteiro.

A impressão de uma linha em branco é dada através de um comando **writeln** como abaixo:

```
writeln ;
```

## Exemplo

```
Program PascalZIM ;
var
  arq: text;
begin
  assign( arq,'teste.txt' ) ;
  rewrite( arq ) ;
  writeln;                                     { Impressão na tela }
  writeln( 1:10, 2:20, 3:30 ) ;
  writeln( 'a':10, 'b':20, 'c':30 ) ;
  writeln( 'asd':10, 'bnm':20, 'cvb':30 ) ;
  writeln( 2.1:10, 3.2:20, 4.3:30 ) ;
  writeln( 2.1:10:2, 3.2:20:3, 4.3:30:4 ) ;
  writeln;                                     { Impressão em arquivo }
  writeln( arq, 1:10, 2:20, 3:30 ) ;
  writeln( arq, 'a':10, 'b':20, 'c':30 ) ;
  writeln( arq, 'asd':10, 'bnm':20, 'cvb':30 ) ;
  writeln( arq, 2.1:10, 3.2:20, 4.3:30 ) ;
  writeln( arq, 2.1:10:2, 3.2:20:3, 4.3:30:4 ) ;
```

```
close( arg ) ;  
End.
```

## Comandos Condicionais

Permitem restringir a execução de comandos.

O compilador reconhece os seguintes comandos condicionais:

- [if..then](#)
- [if...then...else](#)
- [case](#)
- [case...else](#)

## if

Possibilita restringir a execução de um conjunto de comandos.

A sintaxe de um comando **if...then** é:

```
if expressão then comando
```

Onde expressão é uma expressão condicional e comando é um comando simples ou um bloco de comandos.

O comando funciona da seguinte forma: se expressão for TRUE, então comando é executado; caso contrário comando não é executado.

### Exemplo

```
if j <> 0 then result := I/J;
```

A sintaxe de um comando **if...then...else** é:

```
if expressão then comando1 else comando2
```

Onde expressão é uma expressão condicional, comando1 e comando2 um comando simples ou um bloco de comandos.

O comando funciona da seguinte forma: se expressão for TRUE, então comando1 é executado; caso contrário, comando2 é executado.

### Exemplo

```
if j = 0 then  
  write( j )  
else  
  write( m );
```

Em uma série de comandos **if** aninhados a cláusula **else** está ligada ao **if** mais próximo no aninhamento.

Uma seqüência de comandos como:

```
if expressão1 then if expressão2 then comando1 else comando2;
```

É reconhecido pelo compilador da seguinte forma:

```
if expressão1 then [ if expressão2 then comando1 else comando2 ];
```

Pode-se utilizar também, no lugar de vários ifs aninhados, um comando [case](#).

## case

Possibilita a escolha de um conjunto de comandos que serão executados, dentre várias alternativas de escolha.

### Sintaxe

```
case selector of  
  lista de constantes : comandos ;  
  lista de constantes : comandos ;  
  ...  
  lista de constantes : comandos ;  
  else comandos ;  
end ;
```

Onde:

- seletor é uma expressão do tipo integer ou char ;
- lista de constantes é uma sequência de constantes do tipo integer ou char, separadas por vírgula (ao invés de uma constante é possível usar um intervalo de constantes, que consiste em duas constantes separadas por um par de pontos)

A cláusula else não é obrigatória, e os comandos associados a essa cláusula serão executados somente se nenhuma outra opção do case foi selecionada ;

### Exemplo

```
Program PascalZIM ;  
  Var  
    opcao : integer ;  
  
  Begin  
    write ( 'Entre com uma opcao: ' );  
    readln ( opcao );  
  
    // escolha da opcao  
    case opcao of  
      1 : writeln( 'Você escolheu a opção 1...' );  
      2 : writeln( 'Você escolheu a opção 2...' );
```

```
    3 : writeln( 'Você escolheu a opção 3...' );  
    else writeln( 'Você escolheu uma opção diferente de 1, 2, 3...' );  
end ;  
End.
```

## Exemplo

```
Program PascalZIM ;  
const  
    opSoma = '+' ;  
    opSubtracao = '-' ;  
    opProduto = '*' ;  
    opDivisao = '/' ;  
  
Var  
    opcao : char ;  
  
Begin  
    write ( 'Entre com um operador: ' );  
    readln ( opcao );  
  
    // escolha da opcao  
    case opcao of  
        opSoma : writeln( 'Você escolheu soma... ' );  
        opSubtracao : writeln( 'Você escolheu subtracao...' );  
        opProduto : writeln( 'Você escolheu produto...' );  
        opDivisao : writeln( 'Você escolheu divisao...' );  
    end ;  
End.
```

## Exemplo

```
Program PascalZIM ;  
Var  
    opcao : integer ;  
  
Begin  
    write ( 'Entre com uma opcao: ' );  
    readln ( opcao );  
  
    // escolha da opcao  
    case opcao of  
        1, 2 : writeln( 'Você escolheu a opção 1 ou 2...' );  
        3 : writeln( 'Você escolheu a opção 3...' );  
        else writeln( 'Você escolheu uma opção diferente de 1, 2, 3...' );  
    end ;
```

End.

## Exemplo

```
Program PascalZIM ;
  Var
    c: char;

  Begin
    write( 'Digite um caractere: ' );
    readln( c );

    case c of
      'A'..'Z', 'a'..'z': writeln( '=> Você digitou uma letra!' );
      '0'..'9':          writeln( '=> Você digitou um dígito!' );
      '+', '-', '*', '/': writeln( '=> Você digitou um operador!' );
    else
      writeln( '=> Você digitou um caractere!' );
    end;
  End.
```

## ComandosRepeticao

Os comandos de repetição permitem que seja repetida a execução de um conjunto de comandos.

Os comandos de repetição definidos pelo compilador são os seguintes:

- [Repeat](#)
- [While](#)
- [For](#)

Os comandos de desvio que podem ser utilizados nestes comandos são:

- [Break](#)
- [Continue](#)

## repeat

O comando **repeat** executa repetidamente uma sequência de comandos até que uma dada condição, resultantes da avaliação de uma expressão booleana, seja *verdadeira*.

### Sintaxe

```
repeat
  comando1 ;
  ...
  comandoN ;
until expressão ;
```

Onde expressão é uma expressão condicional.

Os comandos internos ao **repeat** são executados no mínimo uma vez, pois a condição de parada da repetição é avaliada somente após a primeira repetição.

### Exemplo

```
repeat
  k := i mod j ;
  i := j ;
  j := k ;
until j = 0 ;
```

# while

O comando **while** é semelhante ao comando **repeat**, com a diferença de que a condição para a execução repetida de comandos é avaliada antes da execução de qualquer comando interno da repetição.

Dessa forma, se a condição inicial para o **while** for *falsa*, a sequência de comandos definidos para o **while** não será executada nenhuma vez.

## Sintaxe

```
while expressão do  
  comando
```

Onde expressão é uma expressão lógica e comando pode ser um comando composto.

## Exemplo.

```
while dado[ i ] <> x do i := i + 1;
```

# for

O comando **for**, diferente dos comandos **repeat** e **while**, permite que uma sequência de comandos seja executada um número definido de vezes.

## Sintaxe

```
for contador := valorInicial to valorFinal do  
  comando
```

```
for contador := valorInicial downto valorFinal do  
  comando
```

Onde:

- contador é uma variável do tipo **integer** ( ou **char** )
- valorInicial e valorFinal são expressões do tipo **integer** ( ou do tipo **char** )
- comando pode ser um comando simples ou um comando composto

## Funcionamento

1. O comando **for** armazena na variável contador o valor da expressão correspondente à valorInicial.
2. Se contador é maior (**for...to**) ou menor (**for...downto**) que valorFinal o comando **for** pára de executar. Caso contrário, comando é executado.
3. Após executar comando o valor armazenado em *contador* é incrementando ou decrementando (o **for...to** incrementa, e **for ...downto** decrementa).
4. Volta para o passo 2.

## Exemplo

```
For i:= 2 to 63 do  
  if data[ i ] > max then  
    max := data[ i ] ;
```

## Exemplo

```
For c:= 'a' to 'z' do  
  write( c );
```

## Comandos Tratamento Arquivos

Dentre os comandos usados para tratamento de arquivos estão inclusos comandos para identificação, abertura e fechamento de arquivos.

O uso de arquivos na linguagem Pascal é feito através da definição de um tipo especial, TEXT, que identifica uma variável do tipo arquivo, definida na parte para declaração de variáveis na seção de definição e declaração de dados.

Os comandos para tratamento de arquivos implementados no compilador são os seguintes:

- [Append](#)
- [Assign](#)
- [Close](#)
- [Reset](#)
- [Rewrite](#)

# append

Abre um arquivo já existente para escrita no final.

## Sintaxe

```
append( variavelArquivo ) ;
```

onde variavelArquivo é uma variável definida com o tipo TEXT.

## Exemplo

```
Program PascalZIM;  
  var  
    arq: Text;  
  begin  
    assign( arq, 'TEST.TXT' );  
    rewrite( arq ); // Cria um novo arquivo  
    writeln( arq, 'texto inicial');  
    close( arq ); // Fecha o arquivo, salvando as alteracoes  
efetuadas  
    append( arq ); // Abre o arquivo para adicionar mais texto no  
final  
    writeln( arq, 'mais texto!' );  
    close( arq ); // Fecha o arquivo, salvando as alteracoes  
efetuadas  
  end.
```

## assign

Um arquivo do tipo texto é referenciado dentro de um programa por uma variável do tipo TEXT. As operações de leitura e escrita em arquivo tomam como argumento essa variável.

Assim, para trabalhar com um arquivo texto deve-se criar uma associação entre a variável TEXT (variável arquivo) e o arquivo armazenado. Essa associação é feita através do comando **assign**.

### Sintaxe

```
assign( variavelArquivo , nomeArquivo );
```

Onde:

- variavelArquivo é uma variável do tipo TEXT
- nomeArquivo é uma cadeia de caracteres contendo o nome do arquivo associado, ou ainda uma variável do tipo string

### Exemplo

```
assign ( arq, 'c:\dados.txt' ) ;
```

O nome do arquivo externo pode ser definido por uma variável do tipo **string**, cujo valor pode ser determinado durante a execução do programa.

### Exemplo

```
Program PascalZIM;  
Var  
  nomeExterno: string [15] ;  
  arq: Text ;  
begin  
  readln ( nomeExterno );  
  assign ( arq, nomeExterno ) ;  
end ;
```



## close

Fecha um arquivo.

### Sintaxe

```
close( variavelArquivo );
```

onde variavelArquivo é uma variável definida com o tipo TEXT.

### Exemplo

```
Program PascalZIM;  
  var  
    arq: Text;  
  begin  
    assign( arq, 'TEST.TXT' );  
    rewrite( arq ); // Cria um novo arquivo  
    writeln( arq, 'texto inicial');  
    close( arq ); // Fecha o arquivo, salvando as alteracoes  
efetuadas  
    append( arq ); // Abre o arquivo para adicionar mais texto no  
final  
    writeln( arq, 'mais texto!' );  
    close( arq ); // Fecha o arquivo, salvando as alteracoes  
efetuadas  
  end.
```

## reset

O comando **reset** abre um arquivo já existente, posicionando o cursor de leitura no seu início.

### Sintaxe

```
reset( variavelArquivo );
```

Onde variavelArquivo é uma variável do tipo TEXT.

### Exemplo

```
Program PascalZIM ;
  var
    arq: Text ;
  begin
    assign( arq, 'TEST.TXT' ) ;
    reset( arq ) ; // Abre o arquivo
    writeln( arq, 'texto inicial' ) ;
    close( arq ) ; // Fecha o arquivo, salvando as alteracoes
efetuadas
    append( arq ) ; // Abre o arquivo para adicionar mais texto no
final
    writeln( arq, 'mais texto!' ) ;
    close(arq) ; // Fecha o arquivo, salvando as alteracoes
efetuadas
  end.
```

## rewrite

O comando **Rewrite** é usado para criar um arquivo do tipo texto ou, se ele já existir, para apagá-lo e criar um novo arquivo, posicionando o cursor de leitura no seu início.

### Sintaxe

```
rewrite( variavelArquivo );
```

Onde variavelArquivo é uma variável do tipo TEXT.

### Exemplo

```
Program PascalZIM;  
  var  
    arq: Text;  
  begin  
    assign( arq, 'TEST.TXT' );  
    rewrite( arq ); // Cria um novo arquivo  
    writeln( arq, 'texto inicial');  
    close( arq ); // Fecha o arquivo, salvando as alteracoes  
efetuadas  
    append( arq ); // Abre o arquivo para adicionar mais texto no  
final  
    writeln( arq, 'mais texto!' );  
    close( arq ); // Fecha o arquivo, salvando as alteracoes  
efetuadas  
  end.
```

## Comandos Auxiliares

O compilador reconhece os seguintes comandos *Pascal*:

- [Break](#)
- [Clrscr](#)
- [Continue](#)
- [Dec](#)
- [Delay](#)
- [Delete](#)
- [Exit](#)
- [Gotoxy](#)
- [Inc](#)
- [Insert](#)
- [Readkey](#)
- [Randomize](#)
- [Str](#)
- [TextBackground](#)
- [Textcolor](#)
- [Val](#)

# break

Usado para forçar a saída de uma estrutura de repetição (while, for, repeat).

## Sintaxe

```
break ;
```

Onde:

- O comando deve estar dentro do corpo de uma estrutura de repetição.
- O próximo comando a ser executado após o break é o comando que segue a estrutura de repetição.

## Exemplo

```
Program PascalZIM;
var
  contador: integer;
begin
  contador := 1;

  { Repetição que é executada 5 vezes }

  while ( true ) do
    begin
      writeln( 'Contador vale:' , contador );
      contador := contador + 1;
      if( contador > 5 ) then
        break
      else
        continue;
    end ;

  { Impressão de uma mensagem após sair da repetição }

  writeln( 'Agora estou fora do while!' );
end.
```



# clrscr

Limpa a tela de impressão.

## Sintaxe

```
clrscr ;
```

## Exemplo

```
Program PascalZIM;  
begin  
  clrscr;  
  writeln( 'Olá, mundo.' );  
end.
```

## continue

Usado para desviar a execução dos comandos de uma estrutura de repetição(while, for, repeat) para a avaliação da condição de loop.

### Sintaxe

```
break ;
```

Onde:

- O comando deve estar dentro do corpo de uma estrutura de repetição.
- Após a execução do comando a repetição pode parar (se a condição de loop assim indicar) ou prosseguir com a execução do primeiro comando da repetição.

### Exemplo

```
Program PascalZIM;
var
  contador: integer;
begin
  contador := 1;

  { Repetição que é executada 5 vezes }

  while ( true ) do
    begin
      writeln( 'Contador vale:' , contador );
      contador := contador + 1;
      if( contador > 5 ) then
        break
      else
        continue;
    end ;

  { Impressão de uma mensagem após sair da repetição }

  writeln( 'Agora estou fora do while!' );
end.
```

---

Created with the Freeware Edition of HelpNDoc: [Single source CHM, PDF, DOC and HTML Help creation](#)

## dec

Diminui de um o valor armazenado em uma variável.

### Sintaxe

```
dec( variável );
```

Onde variável tem o tipo inteiro ou char.

### Exemplo

```
Program PascalZIM;  
var  
  intVar: integer;  
begin  
  intVar := 10;  
  dec( intVar ); { equivalente a intVar:= intVar - 1 }  
end.
```

## delay

Suspende a execução do programa durante X milissegundos.

### Sintaxe

```
delay( expressãoAritmética ) ;
```

Onde expressãoAritmética é uma expressão do tipo integer que indica, em milissegundos, quanto tempo a execução do programa será suspensa.

### Exemplo

```
Program PascalZIM;  
  var  
    miliSegundos: integer;  
  begin  
    write( 'Quanto tempo (em milissegundos) o programa ficará inativo? '  
);  
    readln( miliSegundos );  
    writeln( 'Parando a execução por um tempo... ' );  
    delay( miliSegundos );  
    writeln( 'De volta à ativa! ' );  
  end.
```

# Delete

Usado para remover parte de uma cadeia.

## Sintaxe

```
delete( variável, posInício, quantos ) ;
```

Onde:

- variável é uma variável do tipo **string**.
- posInício é uma expressão do tipo **integer**.
- quantos é uma expressão do tipo **integer**.

## Funcionamento

- O comando remove quantos caracteres da cadeia armazenada em variável, começando da posição posInício.
- A posição do primeiro caractere da cadeia é 1.
- Se posInício é menor ou igual a zero, nenhum caractere é removido da cadeia.
- Se quantos é menor ou igual a zero, nenhum caractere é removido da cadeia.
- Se posInício é maior que o tamanho da cadeia nenhum caractere é removido da cadeia.
- Se a soma de posInício e quantos é maior que o tamanho da cadeia, então quantos é assumido como igual ao tamanho da cadeia -  $\text{posInício} + 1$ .

## Exemplo

Assumindo que a variável *cadeia* armazena "1234567":

- Ao executar o comando `delete(cadeia, 3, 2)` a variável *cadeia* fica armazenando 12567

- Ao executar o comando `delete(cadeia, 1, 3)` a variável *cadeia* fica armazenando 4567
- Ao executar o comando `delete(cadeia, 5, 10)` a variável *cadeia* fica armazenando 1234
- Ao executar o comando `delete(cadeia, 7, 3)` a variável *cadeia* fica armazenando 123456
- Ao executar o comando `delete(cadeia, -3, 3)` a variável *cadeia* fica armazenando 1234567
- Ao executar o comando `delete(cadeia, 0, 3)` a variável *cadeia* fica armazenando 1234567
- Ao executar o comando `delete(cadeia, 7, 0)` a variável *cadeia* fica armazenando 1234567
- Ao executar o comando `delete(cadeia, 5, -2)` a variável *cadeia* fica armazenando 1234567
- Ao executar o comando `delete(cadeia, 9, 5)` a variável *cadeia* fica armazenando 1234567

## Exemplo

```

Program Pzim ;
var cadeia: string ;
Begin
    cadeia := '1234567' ;
    writeln('Valor de cadeia: ', cadeia) ;
    delete(cadeia, 3, 4);
    writeln('Depois do delete: ', cadeia) ; // Mostra 127
End.

```

# exit

Termina a execução do procedimento, função ou programa.

## Sintaxe

```
exit ;
```

## Funcionamento

- Se o comando aparecer dentro de um procedure, o procedimento será encerrado.
- Se o comando aparecer dentro de uma função, a função será encerrada.
- Se o comando aparecer dentro do programa principal, o programa será encerrado.

## Exemplo

```
Program PascalZIM;
```

```
procedure proc ;
```

```
begin
```

```
  writeln('abc'); // Mostra abc
```

```
  exit ;         // Sai do procedure
```

```
  writeln('def'); // Nunca chega aqui
```

```
end;
```

```
function func : integer ;
```

```
begin
```

```
  writeln('ghi'); // Mostra ghi
```

```
  func := 5 ;    // 5 é o valor retornado
```

```
  exit ;        // Sai da funcao
```

```
  writeln('jkl'); // Nunca chega aqui
```

```
end;
```

```
Begin
```

```
  proc ;        // Mostra abc
```

```
  writeln(func) ; // Mostra ghi, depois 5
```

```
  exit;         // Sai do programa
```

```
  writeln('nada'); // Nunca chega aqui
```

```
End.
```



---

Created with the Freeware Edition of HelpNDoc: [Free CHM Help documentation generator](#)

## gotoxy

O comando **gotoxy** define a posição do cursor do teclado na tela de console.

Essa tela possui várias linhas, e cada linha possui um conjunto de colunas.

Pensando em termos de eixos cartesianos, as colunas correspondem ao eixo x, e as linhas ao eixo y.

As linhas aumentam de cima para baixo na tela, enquanto as colunas aumentam da esquerda para a direita

### Sintaxe

```
gotoxy( coluna , linha ) ;
```

Onde coluna e linha tem o tipo inteiro.

### Exemplo

```
Program PascalZIM;
```

```
Begin
```

```
    gotoxy(10,2);  
    textcolor( lightcyan );  
    textbackground( red );  
    write('Olá, mundo!');
```

```
End.
```

## inc

Aumenta de um o valor armazenado em uma variável.

### Sintaxe

```
inc( variável );
```

Onde variável tem o tipo inteiro ou char.

### Exemplo

```
Program PascalZIM ;  
  var  
    intVar: integer ;  
  begin  
    inc( intVar ); { Equivalente a IntVar := IntVar + 1 }  
  end.
```

## Insert

Usado para adicionar uma subcadeia a uma cadeia.

### Sintaxe

```
insert( subcadeia, cadeia, posInicio ) ;
```

Onde:

- subcadeia é expressão do tipo **string**.
- cadeia é uma variável do tipo **string**.
- posInicio é uma expressão do tipo **integer**.

### Funcionamento

- O comando adiciona subcadeia em cadeia, na posição posInicio.
- A posição do primeiro caractere de cadeia é 1.
- Se posInicio é menor ou igual a 1, o comando adiciona subcadeia no início de cadeia.
- Se posInicio é maior que o tamanho da cadeia, o comando adiciona subcadeia no fim de cadeia.
- Se a cadeia resultante tem mais de 255 caracteres, ela é truncada para 255 caracteres.

### Exemplo

Assumindo que a variável *cadeia* armazena "1234567":

- Ao executar o comando `insert('abcd', cadeia, 2)` a variável *cadeia* fica armazenando `1abcd234567`
- Ao executar o comando `insert('abcd', cadeia, 7)` a variável *cadeia* fica armazenando `123456abcd7`
- Ao executar o comando `insert('abcd', cadeia, 1)` a variável *cadeia* fica

armazenandoabcd1234567

- Ao executar o comando `insert('abcd', cadeia, -1)` a variável *cadeia* fica armazenandoabcd1234567
- Ao executar o comando `insert('abcd', cadeia, 8)` a variável *cadeia* fica armazenando1234567abcd

## Exemplo

```
Program Pzim ;  
var cadeia: string ;  
Begin  
    cadeia := '1234567' ;  
    writeln('Valor de cadeia: ', cadeia) ;  
    insert('abcd', cadeia, 4);  
    writeln('Depois do insert: ', cadeia) ; // Mostra 123abcd4567  
End.
```

# randomize

Inicializa o gerador de números randômicos do compilador.

## Sintaxe

```
randomize;
```

## Exemplo

```
Program PascalZIM ;  
var  
  i: integer ;  
begin  
  randomize;  
  repeat  
    i:= i + 1;  
    writeln ( random(1000) );  
  
  until i>10 ;  
end.
```

## str

Usado para converter uma expressão numérica em uma cadeia.

### Sintaxe

```
str( expressãoAritmética , variável ) ;
```

Onde:

- expressãoAritmética é um expressão do do tipo **integer**.
- variável é uma variável do tipo **string**.

### Funcionamento

variável receberá o valor proveniente da conversão.

### Exemplo

```
Program PascalZIM ;  
var s: string[2] ;  
Begin  
    str( 26+3, s );  
    writeln( s );  
End.
```

## textbackground

O comando **textbackground** define a cor de fundo usada na impressão de textos.

### Sintaxe

```
textbackground( listaDeCores ) ;
```

Onde listaDeCores pode ser uma constante inteira ou qualquer uma dentre as cores seguintes:

- o BLUE
- o GREEN
- o CYAN
- o RED
- o MAGENTA
- o BROWN
- o LIGHTGRAY
- o DARKGRAY
- o LIGHTBLUE
- o LIGHTGREEN
- o LIGHTCYAN
- o LIGHRED
- o LIGHMAGENTA
- o YELLOW
- o WHITE
- o BLACK

Exemplo. Pode ser utilizada ainda uma combinação das cores, como em :

```
textbackground( RED + BLUE ) ;
```

Exemplo. Programa que move o cursor do teclado para a linha 15, coluna 10, e imprime "Olá, mundo!" na tela.

**Program** PascalZIM;

**Begin**

```
gotoxy(15,10);  
textcolor( lightcyan );  
textbackground( red );  
write('Olá, mundo!');
```

**End.**

## textcolor

O comando **textcolor** define a cor da fonte usada para impressão de texto na tela.

### Sintaxe

```
textcolor( listaDeCores ) ;
```

Onde listaDeCores pode ser uma constante inteira ou qualquer uma dentre as cores seguintes:

- BLUE
- GREEN
- CYAN
- RED
- MAGENTA
- BROWN
- LIGHTGRAY
- DARKGRAY
- LIGHTBLUE
- LIGHTGREEN
- LIGHTCYAN
- LIGHRED
- LIGHMAGENTA
- YELLOW
- WHITE
- BLACK

Exemplo. Pode ser utilizada uma combinação de cores, como em

```
textcolor ( RED + BLUE ) ;
```

Exemplo. Programa que move o cursor do teclado para a linha 15, coluna 10, e imprime "Olá, mundo!" na tela.

**Program** PascalZIM;

**Begin**

```
gotoxy(15,10);  
textcolor( lightcyan );  
textbackground( red );  
write('Olá, mundo!');
```

**End.**

# val

Usado para converter uma cadeia de caracteres em um inteiro ou real.

## Sintaxe

```
val( expressãoLiteral , variável , codigoErro ) ;
```

Onde:

- expressãoLiteral é uma cadeia de caracteres ou uma expressão envolvendo a concatenação de várias cadeias.
- variável é uma variável do tipo **integer** ou **real**.
- codigoErro é uma variável do tipo **integer**.

## Funcionamento

- Se a cadeia de caracteres puder ser convertida, variável receberá o valor proveniente da conversão, e codigoErro armazenará o valor zero.
- Se a cadeia de caracteres não puder ser convertida, variável receberá o valor zero, e codigoErro armazenará a posição na cadeia em que foi encontrado um caractere inválido.

## Exemplo

- A conversão da cadeia "123" armazena em variável o valor 123 e armazena em codigoErro o valor 0.
- A conversão da cadeia "abc" armazena em variável o valor 0 e armazena em codigoErro o valor 1.
- A conversão da cadeia "123v5" armazena em variável o valor 0 e armazena em codigoErro o valor 4.

## Exemplo

```
Program PascalZIM;
var
  cadeia: string;
  nro, codigoErro: integer;
begin
  write( 'Digite um número inteiro: ' );
  readln( cadeia );
  val( cadeia, nro, codigoErro );
  if ( codigoErro = 0 ) then
    writeln( 'O número lido e convertido foi: ' , nro )
  else
    writeln( 'Inteiro inválido, e o código de erro foi: ' ,
codigoErro );
end.
```

---

Created with the Freeware Edition of HelpNDoc: [Easy to use tool to create HTML Help files and Help web sites](#)

## Subprogramas

Subprogramas são partes de um programa que contém um cabeçalho, uma seção de definição e declaração de dados e uma seção de comandos.

Os subprogramas são definidos na seção de definição e declaração de dados, e podem ser de dois tipos:

- [Procedimentos](#)
- [Funções](#)

A diferença essencial entre *funções* e *procedimentos* é o fato de que as *funções* retornam valores, enquanto os procedimentos não. O valor retornado por uma função é qualquer um dos tipos primitivos **char**, **integer**, **boolean**, **real** ou **string**.

A ativação de um subprograma é feita através de uma *chamada* ao subprograma. Quando um subprograma é *chamado* uma sequência de comandos definida na *seção de comandos* do subprograma é executada, após o qual a execução do programa retorna para instrução seguinte à chamada do subprograma. Um subprograma é chamado através do nome que o define.

Os subprogramas podem ser embutidos; isto é, um subprograma pode ser definido dentro do bloco de declarações de um outro subprograma. Um subprograma embutido pode ser chamado somente pelo subprograma que o contém, sendo visível somente para o subprograma que o contém.

A chamada a um procedimento é reconhecida pelo compilador como um comando, enquanto que uma chamada a uma função é reconhecida como uma expressão.

## Procedimentos e Funções

A declaração de procedimentos e funções difere apenas no cabeçalho. O cabeçalho de um procedimento segue a seguinte sintaxe:

### Sintaxe

```
Procedure NomeProcedimento ;
```

Onde NomeProcedimento identifica o procedimento

O cabeçalho de uma função segue a seguinte sintaxe:

### Sintaxe

```
Function NomeFunção : tipo ;
```

Onde:

- NomeFunção identifica a função
- tipo define o tipo do dado retornado pela função, que pode ser um dos tipos primitivos **char**, **integer**, **boolean**, **real** ou **string**.

A seção de definição e declaração de dados segue o cabeçalho do subprograma, e é o local onde são [definidas as constantes](#) e [tipos](#) passíveis de uso. Também nessa seção são [declaradas as variáveis](#) locais ao subprograma, e definidas as funções e procedimentos que podem vir a serem utilizados pelo subprograma.

A [seção de comandos](#) segue a seção de definição e declaração de dados. É iniciada com a palavra reservada **Begin** e terminada com a palavra reservada **End**, seguida de um ponto e vírgula. Entre as palavras **Begin** e **End** são colocados os comandos da função.

As funções retornam dados através de uma atribuição ao identificador da função de um valor a ser retornado pela função, em alguma parte da [seção de comandos](#) da função.

Funções e procedimentos podem receber [parâmetros](#), e podem ou não serem [recursivos](#).

## Regras de Escopo

As regras de escopo, definidas para um programa escrito na linguagem Pascal, são as seguintes:

- Um identificador definido na *seção de definição e declaração de dados* do programa principal é acessível por qualquer subprograma;
- Um identificador definido na *seção de definição e declaração de dados* de um subprograma é acessível na *seção de comandos* do subprograma na qual foi definido e também na *seção de comandos* de todos subprogramas declarados na sua *seção de definição e declaração de dados*, a menos que esse identificador seja redefinido no subprograma de mais baixo nível na escala hierárquica..
- Os identificadores definidos em um subprograma não existem nem antes nem depois da chamada àquele subprograma e, por isso, não podem ser referenciados em tais momentos.

## Parâmetros

Um subprograma pode receber parâmetros. A definição dos parâmetros passados a um subprograma deve ser especificada no cabeçalho do subprograma, dentro de parênteses.

Os parâmetros podem ter qualquer um dos [tipos predefinidos](#) da linguagem Pascal ( dentre os tipos primitivos implementados no compilador ) ou ainda um tipo que pode ser um dentre os definidos pelo usuário.

A sintaxe do cabeçalho de uma função contendo  $n$  parâmetros é dada, genericamente, por::

```
Function identificador( parâmetro1: tipo ; parâmetro2: tipo ; ... ;  
parâmetron : tipo ) : tipo;
```

A passagem de parâmetros para a função pode ser de dois tipos, a saber:

- Passagem por valor
- Passagem por referência

No primeiro caso o parâmetro assume o valor passado como argumento pela rotina de chamada, e no segundo caso o parâmetro assume o endereço da variável passada como argumento pela rotina de chamada.

A passagem por referência é diferenciada da passagem por valor pela presença da palavra reservada **var** antes do nome identificador do parâmetro.

Exemplo. Dado o seguinte procedimento:

```
Procedure exemplo( var parametroPassadoPorReferencia : integer );
```

Esse procedimento poderia ser chamado através de um comando tal

como:

```
exemplo ( x ) ;
```

onde x é uma variável ( ou constante ) do tipo inteiro.

## Funções Recursivas

Uma função pode chamar a si mesma de dentro de sua própria *seção de comandos*. Quando isto é feito, a função é denominada *função recursiva*. O uso de *funções recursivas* consegue fornecer soluções elegantes para certos tipos de programas, como mostrado no exemplo abaixo, que calcula, para um número inteiro n, seu fatorial:

```
function fatorial (n :integer ) : integer ;  
begin  
  if n > 1 then  
    fatorial := n * fatorial (n-1)  
  else  
    fatorial := 1;  
end;
```

## Funções Pré-Definidas

O compilador implementa as seguintes funções:

- [Abs](#)
- [ArcTan](#)
- [Chr](#)
- [Concat](#)
- [Copy](#)
- [Cos](#)
- [Eof](#)
- [Eoln](#)
- [Exp](#)
- [Frac](#)
- [Int](#)
- [Keypressed](#)
- [Length](#)
- [Ln](#)
- [Odd](#)
- [Ord](#)
- [Pos](#)
- [Pred](#)
- [Random](#)
- [Readkey](#)
- [Round](#)
- [Sin](#)
- [Sqr](#)
- [Sqrt](#)
- [Succ](#)
- [Trunc](#)
- [Uppcase](#)

# abs

Retorna o valor absoluto de um argumento numérico.

## Sintaxe

```
function abs ( x : < integer, real > ): < integer, real > ;
```

## Exemplo

**Program** PascalZIM;

```
var  
  r: Real;  
  i: Integer;  
begin  
  r := abs( -2.3 );    // r recebe 2.3  
  i := abs( -157 );   // i recebe 157  
end.
```

## arctan

Retorna o arco tangente do argumento numérico ( onde x é um ângulo, em radianos).

### Sintaxe

```
function arcTan ( x: real ): real;
```

### Exemplo

```
Program PascalZIM;  
var  
  r: real;  
begin  
  r := arcTan( 3.14 );  
end.
```

## chr

Recebe como parâmetro um inteiro e retorna o caracter ASC II correspondente ao código identificado com esse inteiro.

### Sintaxe

```
function chr( x: integer ): char;
```

### Exemplo

```
Program PascalZIM;  
var  
  i: integer;  
begin  
  for i := 32 to 126 do  
    write( chr(I) );  
end.
```

## concat

Concatena uma sequencia de cadeias de caracteres.

### Sintaxe

```
function concat ( s1, s2, s3, ... : string ): string ;
```

Onde s1, s2, s3 são expressões do tipo string. As reticências indicam que mais de uma expressão pode ser informada para a função concat.

O número mínimo de expressões informadas para a função concat é um.

As expressões informadas podem também ter o tipo char.

### Exemplo

```
Program PascalZIM;  
  var  
    s1, s2, cadeia: string ;  
    vetor: array[1..4] of char ;  
  begin  
    // Exibe Compilador Pascalzim  
    s1 := 'Compilador';  
    s2 := 'Pascalzim';  
    cadeia := concat(s1, ' ', s2) ;  
    writeln( cadeia );  
  
    // Exibe pzim  
    vetor[1] := 'p' ;  
    vetor[2] := 'z' ;  
    vetor[3] := 'i' ;  
    vetor[4] := 'm' ;  
    writeln( concat(vetor[1],vetor[2],vetor[3],vetor[4]));  
  end.
```

## copy

Retorna parte de uma cadeia de caracteres.

### Sintaxe

```
copy( cadeia, posInicio, quantidade ) : string ;
```

Onde:

- cadeia é uma expressão do tipo **string**.
- posInicio é uma expressão do tipo **integer**.
- quantidade é uma expressão do tipo **integer**.

### Funcionamento

- Retorna uma subcadeia de cadeia, que começa na posição dada por posInicio. Quantidade denota a quantidade de caracteres que serão retornados a partir da posição informada.
- O primeiro caractere da cadeia está armazenado na posição 1
- Se quantidade for menor ou igual a zero, será retornada uma cadeia vazia.
- Se posInicio for maior que o tamanho da cadeia, será retornada uma cadeia vazia.
- Se posInicio for menor ou igual a zero, será assumido que posInicio corresponde ao início da cadeia.
- Se a soma de posInicio e quantidade for maior que o tamanho da cadeia, retorna a subcadeia de cadeia que começa em posInicio.

### Exemplo

- copy( 'abcdef', 3, 4 ) produz como resultado a cadeia 'cdef'
- copy( 'abcdef', 3, -4 ) produz como resultado a cadeia vazia
- copy( 'abcdef', 30, 4 ) produz como resultado a cadeia vazia
- copy( 'abcdef', -3, 4 ) produz como resultado a cadeia 'abcd'
- copy( 'abcdef', 3, 20 ) produz como resultado a cadeia 'cdef'

## Exemplo

**Program** PascalZIM;

**var**

cadeia: string ;

**begin**

cadeia := 'abcdef' ;

writeln( copy(cadeia, 3, 4) ) ; // Exibe cdef

writeln( copy(cadeia, -3, 4) ) ; // Exibe abcd

writeln( copy(cadeia, 30, 4) ) ; // Exibe cadeia vazia

writeln( copy(cadeia, 4, -2) ) ; // Exibe cadeia vazia

**end.**

## COS

Retorna o coseno do argumento ( x é um ângulo, em radianos ).

### Sintaxe

```
function cos ( x: real): real ;
```

### Exemplo

```
Program PascalZIM;  
var  
  r: real;  
begin  
  r := cos( 3.14 ); // Imprime o cosseno de pi  
end.
```

## eof

Usada para verificar se o final de um arquivo foi alcançado durante uma leitura de valores.

### Sintaxe

```
function eof ( var arquivo: text ): boolean;
```

A função recebe como argumento uma variável do tipo TEXT, retornando **true** se o cursor de leitura do arquivo referenciado por F se encontra no seu fim, **false** em caso contrário.

### Exemplo

```
var
  arq: Text ;
  caractere: char ;
begin
  assign( arq, 'teste.pas' );
  reset( arq );
  while not eof( arq ) do
    begin
      read( arq, caractere );
      write( caractere );
    end;
end.
```

## eoln

Usada para verificar se o final de uma linha em um arquivo do tipo texto foi alcançado durante uma leitura de valores.

### Sintaxe

```
function eoln ( var arquivo: text ): boolean;
```

A função recebe como argumento uma variável do tipo TEXT, retornando **true** se o cursor de leitura do arquivo referenciado por F se encontra no fim.de uma linha, **false** em caso contrário.

### Exemplo

```
var  
  arq: Text ;  
  caractere: char ;  
begin  
  assign( arq, 'teste.pas' );  
  reset( arq );  
  while not eoln( arq ) do  
  begin  
    read( arq, caractere );  
    write( caractere );  
  end;  
end.
```

## exp

Retorna o exponencial do argumento.

### Sintaxe

```
function exp ( x: real ) : real;
```

### Exemplo

#### **Program** PascalZIM;

```
  begin  
    writeln( 'e = ', exp(1.0) );  
end.
```

# frac

Retorna a parte fracionária de um valor numérico.

## Sintaxe

```
frac( valor ): real ;
```

Onde:

- valor é uma expressão do tipo integer ou real

## Exemplo

```
Program PascalZIM;  
begin  
  writeln( int(12.34) ) ; // Mostra 12.00  
  writeln( frac(12.34) ) ; // Mostra 0.34  
  
  writeln( int(12) ) ; // Mostra 12.00  
  writeln( frac(12) ) ; // Mostra 0.00  
end.
```

# int

Retorna a parte inteira de um valor numérico.

## Sintaxe

```
int( valor ): real ;
```

Onde:

- valor é uma expressão do tipo integer ou real

## Exemplo

```
Program PascalZIM;  
begin  
  writeln( int(12.34) ) ; // Mostra 12.00  
  writeln( frac(12.34) ) ; // Mostra 0.34  
  
  writeln( int(12) ) ; // Mostra 12.00  
  writeln( frac(12) ) ; // Mostra 0.00  
end.
```

# keypressed

Verifica se foi pressionada alguma tecla.

## Sintaxe

```
keypressed : boolean ;
```

## Exemplo

```
Program PascalZIM ;  
Begin  
  while not keypressed do  
    Begin  
      write('x');  
    End;  
End.
```

## length

Retorna o comprimento de uma cadeia de caracteres.

### Sintaxe

```
length( expressãoLiteral ): integer ;
```

Onde expressãoLiteral é uma cadeia de caracteres ou uma expressão envolvendo a concatenação de várias cadeias.

### Exemplo

```
Program PascalZIM;  
  var  
    s: string;  
  begin  
    write( 'Digite uma cadeia: ' );  
    readln( s );  
    writeln( ' O comprimento da cadeia lida = ', length( s ));  
  end.
```

# ln

Retorna o logaritmo neperiano do argumento.

## Sintaxe

```
function ln ( x: real ): real;
```

## Exemplo

```
Program PascalZIM ;  
var  
    e: real;  
begin  
    e := exp( 1.0 );  
    writeln( 'ln(e) = ', ln( e ) );  
end.
```

## odd

Verifica a paridade do argumento, retornando **true** se o argumento é ímpar, **false** em caso contrário.

### Sintaxe

```
function odd ( x: integer ): boolean ;
```

### Exemplo

```
Program PascalZIM ;  
  begin  
    if odd( 5 ) then  
      writeln( '5 é impar!' )  
    else  
      writeln( '5 não é impar...' );  
  end.
```

## ord

Recebe como parâmetro um caractere e retorna o inteiro correspondente ao código ASCII referente ao caractere.

### Sintaxe

```
function Ord ( X : char ): integer ;
```

### Exemplo

```
Program PascalZIM;  
begin  
    writeln( 'O código ASCII para "c" = ', ord('c'), ' decimal' );  
end.
```

## pos

Retorna a posição de uma subcadeia dentro de uma cadeia de caracteres.

### Sintaxe

```
pos( subcadeia , cadeia ): integer ;
```

Onde:

- subcadeia é a cadeia que será utilizada na busca
- cadeia é a cadeia onde será procurada a subcadeia

### Funcionamento

- Se a subcadeia não for encontrada na cadeia, a função pos retorna zero.
- A posição do primeiro caractere da cadeia é um.

### Exemplo

pos( 'zim', 'Pascalzim' ) produz como resultado 7

pos( 'zIm', 'Pascalzim' ) produz como resultado 0

### Exemplo

```
Program PascalZIM;  
begin  
  writeln( pos('zim', 'Pascalzim') ); // Mostra 7  
  writeln( pos('zIm', 'Pascalzim') ); // Mostra 0  
end.
```



## pred

Retorna o número/caractere que antecede o argumento.

### Sintaxe

`pred( variável ) : integer ou char ;`

onde variável tem o tipo inteiro ou char.

### Exemplo

```
Program PascalZIM ;  
  begin  
    writeln( 'O predecessor de 5 = ', Pred(5) );  
    writeln( 'O sucessor de 10 = ', Succ(10) );  
  end.
```

## random

Recebe como parâmetro um inteiro  $x$  e retorna um número  $n$  no intervalo  $0 \leq n < x$ .

### Sintaxe

```
random( x ): integer;
```

### Exemplo

```
Program PascalZIM ;
var
  i: integer ;
begin
  randomize;
  repeat

    i:= i + 1;
    writeln ( random(1000) );

  until i>10 ;
end.
```

## readkey

Solicita a leitura de um caracter do teclado. Pode ser utilizado como um comando ou como uma função.

### Sintaxe

```
readkey ;
```

### Exemplo

```
Program PascalZIM ;  
begin  
  writeln( 'O programa vai terminar...' );  
  readkey;  
end.
```

Como função, sua sintaxe é:

```
readkey: integer;
```

### Exemplo

```
Program PascalZIM ;  
var  
  umCaractere: char ;  
begin  
  writeln( 'Digite um caracter:' );  
  umCaractere:= readkey;  
  writeln( 'Você digitou: ', umCaractere );  
end.
```

# round

Arredonda um valor real em um valor inteiro.

## Sintaxe

```
function Round ( X: Real ): integer;
```

## Exemplo

**Program** PascalZIM ;

**begin**

```
writeln( 1.4, ' é arredondado para ', round( 1.4 ) ) ;
```

```
writeln( 1.5, ' é arredondado para ', round( 1.5 ) ) ;
```

```
writeln( -1.4, ' é arredondado para ', round( -1.4 ) ) ;
```

```
writeln( -1.5, ' é arredondado para ', round( -1.5 ) ) ;
```

**end.**

# sin

Retorna o seno do argumento ( x é um ângulo, em radianos ).

## Sintaxe

```
function sin ( x: real ): real;
```

## Exemplo

```
Program PascalZIM ;  
  var  
    r: real;  
  begin  
    r := sin( 3.14 ) ;  
    writeln ( 'O seno de Pi = ', r ) ;  
    readln ;  
  end.
```

## sqr

Retorna o quadrado do argumento.

### Sintaxe

```
function sqr ( x : < integer, real > ): < integer, real > ;
```

### Exemplo

```
Program PascalZIM ;  
  begin  
    writeln( 'O quadrado de 5 = ', sqr(5) ) ;  
    writeln( 'A raiz quadrada de 2 = ', sqrt(2.0) ) ;  
  end.
```

# sqrt

Retorna a raiz quadrada do argumento.

## Sintaxe

```
function sqrt ( x: real ): real;
```

## Exemplo

```
Program PascalZIM ;  
  begin  
    writeln( 'O quadrado de 5 = ', sqr(5) ) ;  
    writeln( 'A raiz quadrada de 2 = ', sqrt(2.0) ) ;  
  end.
```

## SUCC

Retorna o número/caractere que sucede o argumento.

### Sintaxe

```
succ( variável ) : < integer ou char >;
```

onde variável tem o tipo inteiro ou char.

### Exemplo

```
Program PascalZIM ;
```

```
begin
```

```
  writeln( 'O predecessor de 5 = ', pred(5) ) ;
```

```
  writeln( 'O sucessor de 10 = ', succ(10) ) ;
```

```
end.
```

# trunc

Trunca um valor real em um valor inteiro.

## Sintaxe

```
function trunc ( x: real ): integer;
```

## Exemplo

```
Program PascalZIM ;
```

```
  begin
```

```
    writeln( 1.4, ' se torna ', trunc( 1.4 ) ) ;      { 1.0 }
```

```
    writeln( 1.5, ' se torna ', trunc( 1.5 ) ) ;      { 1.0 }
```

```
    writeln( -1.4, ' se torna ', trunc( -1.4 ) ) ;    { -1.0 }
```

```
    writeln( -1.5, ' se torna ', trunc( -1.5 ) ) ;    { -1.0 }
```

```
  end.
```

## upcase

Recebe como parâmetro um caractere e retorna sua representação em caixa alta.

### Sintaxe

```
function upcase( c: char ): char;
```

### Exemplo

```
Program PascalZIM;  
var  
  umCaractere: char;  
  cadeia: string;  
  i: integer;  
begin  
  cadeia:= 'Uma frase' ;  
  for i:=0 to length( cadeia ) do  
    write( upcase( cadeia[ i ] ) ) ;  
end.
```

## Tratamento de Overflow

O tratamento de overflow, no compilador, é realizado para constantes inteiras, reais e literais.

O tratamento consiste em verificar se, durante uma determinada operação, uma constante ultrapassa o valor máximo permitido para constantes do tipo em questão.

O intervalo de valores válidos para constantes numéricas é:

- Para constantes reais:  $3.4 * (10^{**-38})$  à  $3.4 * (10^{**+38})$
- Para constantes inteiras: 32767 à -32767.

O tratamento de *overflow* dado às cadeias de caracteres depende do tamanho da cadeia.

## Comentarios

Os comentários são usados dentro de um programa com a finalidade de documentar trechos de código, e não afetam a execução do programa. A definição de um comentário, na linguagem Pascal, é feita com o uso de pares de chaves { } ou (\* \*).

O texto comentado é ignorado pelo compilador.

### Exemplo

```
Program PascalZim; { Esse é meu programa de teste }
Begin
  Write('Olá, mundo!'); { Imprime a mensagem 'Olá, mundo!' }
End.
```

### Exemplo

```
Program PascalZim; (* Esse é meu outro programa de teste *)
Begin
  Write('Olá, mundo!'); (* Imprime a mensagem 'Olá, mundo!' *)
End.
```

O Pascalzim possui ainda um outro tipo de comentário, o comentário de linha. Um comentário de linha é iniciado por // e todos os caracteres que seguem o // na linha são automaticamente ignorados pelo compilador, da mesma forma que em C e Java.

### Exemplo

```
Program PascalZIM;
Begin
  Write( 'Olá, mundo!' ); // Esse é um comentário de linha
End.
```